KCA 401 Computer Science Honours (Part-Time)

Self Study

# 3D APIs in Interactive Real-Time Systems:

## Comparison of OpenGL, Direct3D and Java3D.

Written by: **Richard Dazeley**

Supervised by: **Peter Vamplew**

A report submitted to the
**Faculty of Science and Technology,**

## University of Tasmania

In partial fulfilment of the requirements for the degree of
**Bachelor of Computer Science with Honours**

October 2000

# Contents

# Figures

# Tables

# CHAPTER 1
# Introduction

Since the first display of a few computer-generated lines on a Cathode-ray tube (CRT) over 40 years ago, graphics has progressed rapidly towards the computer generation of detailed images and interactive environments in real time (Angel, 1997). In the last twenty years a number of Application Programmer's Interfaces (APIs) have been developed to provide access to three-dimensional graphics systems. Currently, there are numerous APIs used for many different types of applications. This paper will look at three of these: OpenGL, Direct3D, and one of the newest entrants, Java3D. They will be discussed in relation to their level of versatility, programability, and how innovative they are in introducing new features and furthering the development of 3D-interactive programming.

## *1.1 History - Four Generations of 3D APIs*

The steady advance in hardware technology has facilitated the development of a number of 3D APIs and standards. In the mid-seventies a growing awareness of the need for graphics standards led to the first generation of 3D APIs such as *Core* (short for, *3D Core Graphics System*), produced by an ACM SIGGRAPH Committee in 1977. The 2D component of Core was later expanded and cleaned up to be the first officially standardised graphics specification called the *Graphical Kernel System* (GKS)(Foley, 1996).

During the early eighties, several 3D API standards were developed to cater for the newly emerging workstations. These second generation APIs allowed for networking and basic hardware acceleration (Mohan, 1998). Two of these APIs were later standardised in 1988; GKS-3D and a far more sophisticated and complex system called the *Programmer's Hierarchical Interface Graphics System* (PHIGS). As the name suggests PHIGS supports nested hierarchical groupings of 3D primitives called structures (Foley, 1996). These groupings, referred to as *display lists*, have the advantage of only having to describe a complex object once even if displayed several times. This ability is especially valuable where object data to be

displayed must be transmitted via a low bandwidth network. One disadvantage of the display list is the difficulty in re-specifying an object if it is continuously being updated due to user interaction (Segal, 1994). The latest incarnation of PHIGS is PHIGS+ which is designed with an additional set of features for modern, pseudorealistic rendering of objects on raster display systems (Foley, 1996). However, a major drawback on all these APIs is that they lack support for a number of the more advanced rendering features such as texture mapping (Segal, 1994).

Another API released around this time was PEX, which is an extension to the X Windows system, used for manipulating and drawing 3D objects. While it is based on PHIGS, PEX allows for immediate mode rendering. This means that objects are displayed as they are described rather than first having to complete a display list. While PEX does not support advanced rendering features and is only available to X windows users, its methods used to describe and render objects are similar to those provided by OpenGL (Segal, 1996).

The late 1980s saw the release of the OpenGL standard, based on IRIS GL by Silicon Graphics. It took the high-end 3D graphics market by storm. Like PEX before it, OpenGL offered immediate mode graphics along with enhanced lighting capabilities and advanced features like texture mapping and antialiasing (Mohan, 1998). OpenGL also allowed for hardware acceleration if the hardware supported the precision conformance requirements. This, however, tended to be high-priced hardware (until recently when cheaper OpenGL supported hardware has become available) and so OpenGL tended to be more directed towards CAD and CAE workstations (Microsoft, 1995).

The low-priced, PC market had relied on 2.5D (3D in appearance but using 2D techniques like *Doom* and *Hexen*) and basic 3D (like *Ultima Underworld*) DOS based graphics engines for games (De Goes, 1996). With the advent of Windows 95, Microsoft also released DirectX which included Direct3D with the goal of producing an API that was aimed at providing most of the advanced features of OpenGL specifically for the 3D gaming environment (Microsoft, 1995). Initially, the API was a poor imitation that was limited and difficult to use. Unlike OpenGL though, it has been repeatedly revised and been a lot more innovative in the introduction of new features (anon (a), 1999).

Most recently a fourth generation of APIs has begun to emerge using *scene-graph-based* systems. This flexible tree structure is similar to the display list techniques used in PHIGS and other early APIs but is much more robust. The system incorporates a number of previous ideas such as back-facing polygon removal and BSP trees. These new APIs include extensions to existing APIs such as *OpenGL Optimizer* by SGI and *Fahrenheit* by Microsoft. One problem with these APIs is that the programmer still needs some understanding of the underlying low-level API. However, one completely new API in this group is Java 3D and because it was built using the scene-graph-based system from the outset, it does not require as steep a learning curve, allowing new users easy access into the 3D-development environment (Mohan, 1998).

The table below briefly summarises these four generations and the major APIs discussed earlier. The third generation APIs, OpenGL and DirectX along with the new entrant Java 3D will be discussed in more detail later in this paper.

| API | Hardware | Language | Connections | 3D Graphics | New Users |
|---|---|---|---|---|---|
| Siggraph Core | Raster Terminals<br>Minicomputers | FORTRAN | Serial I/O | 3D Primitives<br>Surfaces | MCAD<br>Companies |
| PHIGS/<br>PEX | Workstations | C | I/O Busses<br>Ethernet | Display Lists<br>Solids<br>Basic 3D Acceleration | MCAE<br>Workstation Users |
| OpenGL/<br>DirectX | Advanced 3D Accelerated Workstations<br>Personal Computers | C++ | Integrated 3D Graphics Systems | Immediate Mode<br>Texture Mapping<br>Advanced Lighting<br>Advanced 3D Hardware | Animation<br>Simulation<br>Game Developers |
| Java 3D/<br>OpenGL Optimizer/<br>Fahrenheit | Java Terminals<br>Network Computers<br>Set-top Boxes<br>Game Consoles | Java<br>C++ | Internet Networks | Scene Graphs<br>Geometry Compression<br>Behaviours<br>Spatial Sound | Game Developers<br>Web Page Designers<br>Information Distributors |

**Table 1.1: Four Generations of 3D APIs (Mohan, 1998)**

## *1.2 API Requirements*

There are a number of fundamental considerations for interactive 3D graphics APIs to address. To render a 3D scene of only modest complexity requires numerous calculations and when used in an interactive application must do this several times per second. Therefore, the API would ideally need to be able to access the capabilities of the graphics hardware (Segal, 1994).

The interface should also provide versatility in the form of allowing the programmer to switch on or off the various rendering features. This allows the programmer to tailor the application's performance to the system it is running on. It also means that during times of high interactive rates, such as adjusting the camera position, performance-degrading combinations of features can be switched off. The camera can then be moved without loss of performance and the viewer will not notice the decrease in detail on a moving image. When the movement is complete the features can be switched back on for the final frame (Segal, 1994).

All APIs come with a large array of functions in their library. Therefore, the functions in each of the APIs discussed in this paper will be divided into six groups according to their functionality:

1. The *primitive functions,* which define low-level or atomic entities that can be displayed.
2. *Attribute functions* that allow operations to be performed on those objects such as colour, shading and texturing.
3. *Viewing functions* that the API provides such as positioning, orienting and selecting the equivalent of a lens.
4. The types of *transformation functions* available for the programmer to manipulate objects.
5. What *input functions* are provided, allowing versatility with various external devices.
6. The *control functions* that allow us to be able to work in a multi-processing, multi-window and networked environment along with initialisation and error handling functions.

## *1.3 Literature*

Three-dimensional programming and APIs have been around long enough for an extensive array of literature to be developed. There are numerous books on how to build 3D systems from the ground up explaining in detail various algorithms used for various effects. Also, each of the major APIs release significant details on their particular systems along with independent books detailing how to use them. There is also a large amount of material published in journals and on the Internet that looks at particular aspects of the various APIs.

Should the reader seek further information on the topics discussed in this paper then the following texts are highly recommended:

- For 3D programming techniques, algorithms and the PHIGS API; see *Computer Graphics: Principles and Practice* by Foley, 1996, or *Computer Graphics: C Version* by Hearn, 1997.
- For more detailed information on the OpenGL API see the *OpenGL Programmer's Guide*, 1993, or the *OpenGL Reference Manual*, 1999.
- For information on Direct X see the *Microsoft Direct X SDK*, or *Inside Direct 3D* by Kovack, 2000.
- For more information on Java 3D see the Java 3D 1.2 API web site at http://java.sun.com/products/java-media/3D/ or *The Java 3D API Specification* by Sowizral, 2000.

## *1.4 Report Outline*

This paper is divided into six chapters: the first chapter provides a brief history of the development of APIs along with an overview of the requirements of a good API. Chapter 2 will look briefly at the basics of 3D to familiarise the reader with some of the concepts involved. Chapters 3, 4 and 5 will investigate OpenGL, Direct3D and Java 3D respectively. They will outline how these APIs work and in addition they will explain and give examples of how to perform basic programming as well as outlining the advantages and disadvantages inherent in each API. The final chapter will briefly overview the main points discussed.

# CHAPTER 2
# 3D Programming Overview

The process of modelling and displaying a 3D scene is far more complex and involves more than just adding a third coordinate to 2D graphics systems. Firstly, an object boundary can be constructed with a combination of various plane and curved surfaces, and sometimes it can specify information about the object's interior. Also, geometric transformations can be more involved in three-dimensional space, as we can rotate an object around an axis with any spatial orientation, whereas rotations in two-dimensional space are always perpendicular to the *xy* axis (Hearn, 1997).

The extra complexity does not only arise from adding an extra dimension but also from the mismatch between the 3D environment and the 2D viewing display device (Foley, 1996). This makes viewing transformations much more complicated because there are many more parameters to select to specify how the scene should be mapped to the display device. Also, to complete rendering non-visible parts of the scene need to be clipped, hidden surfaces removed and surface-rendering algorithms applied (Hearn, 1997).

It is not the intention of this paper to explain these areas in detail, as there are numerous books that do, such as *Computer Graphics: Principles and Practice* by James Foley, 1996. However, this paper will give some basic background material for the purpose of providing a footing for the concepts discussed later.

## *2.1 Coordinate System*

There are a number of coordinate systems that can be used but generally most APIs use either the left-handed or right-handed Cartesian coordinate systems. Systems such as Polar Coordinates are not used due to their computational overhead. Both the left and right-handed systems have the positive x-axis pointing to the right and the positive y-axis pointing up. In the left handed system the z-axis points away from the viewer, whereas it points towards the viewer in the right handed system (see Figure 2.1).

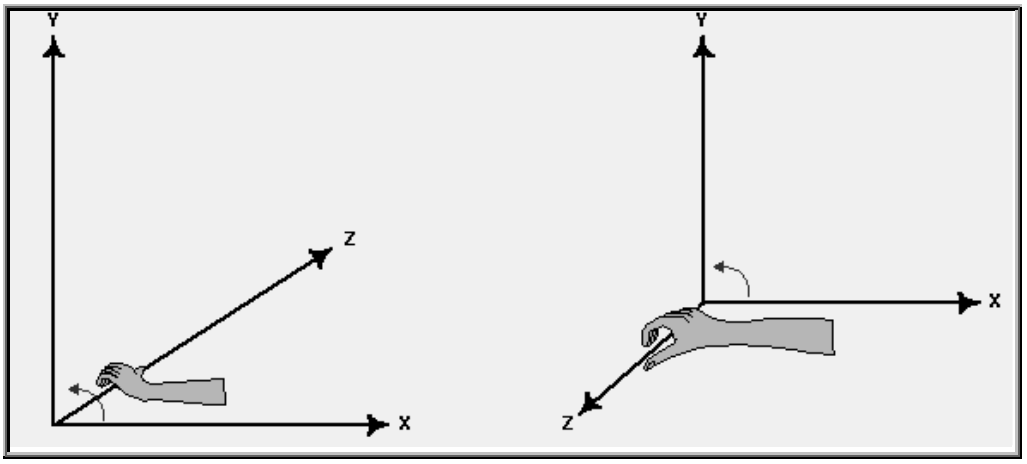


**Figure 2.1: 3D Coordinate Systems (Microsoft, 1999, p3)**

## *2.2 Primitives*

Due to hardware and software limitations the full range of three-dimensional primitives, such as curved surfaces like spheres, are not represented. Currently, for objects to fit graphics hardware they should have the following characteristics:

- The object is described by its surfaces and can otherwise be thought of as hollow.
- The object is specified by a set of vertices in three-dimensions.
- The object is composed of flat convex polygons.

Therefore, two-dimensional flat primitives are required to render three-dimensional objects. In addition curved surfaces need to be approximated. The second condition comes about from the concept of the pipeline architecture for graphics systems and allows faster rendering.

Some systems claim to allow curved surfaces, in contradiction of the third point above. In these systems, however, when a set of vertices are passed to the graphics system that do not all exist on the same plane, the graphics system *tessellates* the polygon into triangles (as triangular polygons can only exist in a single plane). Thus, curved objects are composed of small flat triangles that approximate the curve (Angel, 1997).

## *2.3 Geometric Transformations*

Geometric Transformations are used to change a set of points in three-dimensional space to a new set of points in a uniform manner. This is usually done using a 4x4 matrix. The combination of particular parts of the matrix can perform different types of transformation. This method also allows us to build a single matrix out of a series of transformations using matrix-concatenation (Hearn, 1997).

**Note:** All of the transformations given in this section are using a right-handed coordinate system, transpose matrices for left-handed coordinate system.

### *2.3.1 Translation Transformations*

The translation transformation moves the set of points linearly in space. A point is translated from position P = (x, y, z) to the new position P' = (x', y', z') using the following matrix operation.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

**Figure 2.2: Translation Transformation Matrix (Hearn, 1997, p408)**

The parameters $t_x$, $t_y$ and $t_z$ specify the translation distances of the original point in the coordinate system (Hearn, 1997).

### *2.3.2 Rotation Transformations*

To perform any type of rotation, there are two things that need to be specified: the axis of rotation and the amount of angular rotation. In three-dimensional space a rotation can be specified around any line. The easiest rotation axis to handle are those that are parallel to the coordinate axis. Also, through the combination of a number of coordinate-axis rotations and appropriate translations we can specify any general rotation much more easily than trying to specify it directly. The angle of rotation, usually given in radians, by convention usually produces a counterclockwise rotation when positive (Hearn, 1997).

In the following figures three matrix-templates are given for x, y and z coordinate-axis rotation. In all cases ? is the angle of rotation in radians.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos ? & -\sin ? & 0 \\ 0 & \sin ? & \cos ? & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

**Figure 2.3: X-Axis Rotation Matrix (Hearn, 1997, p411)**

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos ? & 0 & \sin ? & 0 \\ 0 & 1 & 0 & 0 \\ -\sin ? & 0 & \cos ? & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

**Figure 2.4: Y-Axis Rotation Matrix (Hearn, 1997, p412)**

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos ? & -\sin ? & 0 & 0 \\ \sin ? & \cos ? & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

**Figure 2.5: Z-Axis Rotation Matrix (Hearn, 1997, p410)**

### *2.3.3 Scaling Transformations*

Scaling transformations are used to change an object's size and are performed using the following matrix.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

**Figure 2.6: Scaling Transformation Matrix (Hearn, 1997, p421)**

The variable's $s_x$, $s_y$ and $s_z$ must all be positive real values and represent the amount of scaling in each of the coordinate directions. Therefore, if $s_x = s_y = s_z$ then the original dimensions of the object will be maintained. The above matrix performs scaling relative to the coordinate system's origin. A fixed-position scaling must be performed instead, when an object is not at the origin, by using matrix concatenation as shown in the next section (Hearn, 1997).

### *2.3.4 Transform Concatenation*

One of the primary reasons for using matrices to represent transformations is that you can combine the effects of a number of matrices by multiplying them. Therefore, you can rotate and translate an object with a single matrix rather than applying two separate matrices to the graphics hardware. This multiplication (or concatenation) is performed from right to left, where the furthest right is the first transformation to be applied.

For example, to perform a fixed-position scaling of an object there are three steps involved:

1. Translate the fixed point to the origin.
2. Scale the object.
3. Translate the fixed point back to its original position.

If these three transformations are multiplied the result is a single matrix that will perform all the above operations at once as illustrated in the Figure 2.7.

$$\begin{bmatrix} 1 & 0 & 0 & x_f \\ 0 & 1 & 0 & y_f \\ 0 & 0 & 1 & z_f \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_f \\ 0 & 1 & 0 & -y_f \\ 0 & 0 & 1 & -z_f \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & x_f(1-s_x) \\ 0 & s_y & 0 & y_f(1-s_y) \\ 0 & 0 & s_z & z_f(1-s_z) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Figure 2.7: Fixed Position Scaling using Matrix Concatenation (Hearn, 1997)**

## *2.4 Shading*

Having the restriction of only being able to use flat surfaces to represent curved surfaces only produces satisfactory results if the triangles are very small. A large curved surface, however, could have far too many triangles for even the most expensive hardware to render in real time. This problem is solved through making the flat surfaces appear to be curved through the application of shading.

### *2.4.1 Gouraud Shading*

In 1971, Joe-Bob Gouraud introduced an algorithm that uses a technique known as *intensity interpolation*, to make curved surfaces approximated by that polygon appear smooth. It has since become known as Gouraud shading. The algorithm calculates a vector for each vertex (corner) by averaging the normals

of the polygons that share that vertex. Then for each vertex the colour and intensity is then calculated. The angles and distance to the various light-sources gives the lights intensity (De Goes, 1996). Then the colours of the light and the material are combined and used on the polygon, thus giving the vertices colour (Microsoft, 1999).

The last stage of the algorithm is to interpolate these values down the edges and across the scan-lines of the polygons. This is done linearly providing a smooth transition from one value to the next. The algorithm is not much slower than simply using *flat shading* but significantly improves the appearance of a surface (Microsoft, 1999). The main problem with the algorithm is that it fails to detect highlights that fall between vertices (De Goes, 1996).

### *2.4.2 Phong Shading*

Bui-Tong Phong's shading algorithm, also known as *normal-vector interpolation shading*, interpolates the surface normal vector rather than the intensity (Foley, 1996). A polygon is rendered using Phong shading in a three-step process. Like Gouraud shading, the average normal at each vertex is determined. Then, a series of vertex normals are interpolated over the surface of the polygon. Now, the illumination model is applied along each scan line to calculate pixel intensities (Hearn, 1997).

This method means that highlights that occur in between vertices are detected which significantly improves the curved look over Gouraud shading (see figure 2.2), especially when using a *specular reflectance* illumination model. The problem with the algorithm is that the interpolated normal must be normalised every time it is used in the illumination model (Foley, 1996). This greatly increases the number of calculations and is usually unrealistic in most uses, especially in real time interactive environments.

There are a number of approximations to the Phong method that have also been developed that are much faster and still give a reasonable result. Some methods in use are the *Fast Phong shading,* by Bishop and Weimer, which approximates the intensity calculations using a Taylor-series expansion and triangular surface

patches. Another method developed by Duff uses a combination of difference equations and table lookups (Foley, 1996).

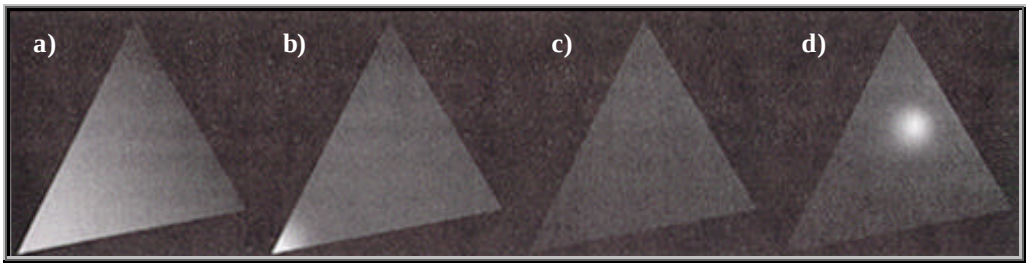


**Figure 2.8: A specular-reflection illumination model used with Gouraud and Phong shading. Highlight at left vertex: (a) Gouraud shading, (b) Phong shading. Highlight falls in polygon interior: (c) Gouraud shading, (b) Phong shading (Foley, 1996, p739).**

## *2.5 Lighting and Materials*

In the real world environment a surface can either emit light through self-emission or reflect light from other surfaces that illuminate it and some may do both. When we look at the surface of an object, the colour we see is the result of multiple light sources and reflective surfaces interacting with the type of the material on the surface (Angel, 1997).

The real physical lighting model can not be calculated in the general case, and approximations such as radiosity and ray tracing are too slow for real-time environments. Consequently, the basic system employed in most APIs involves the placement of light sources of different types and colours in a scene. Each polygon surface is also allocated a material that describes how the polygon will reflect light; the *reflectance model* (Angel, 1997).

### *2.5.1 Light Sources*

There are numerous types of light sources in nature but most scenes can be rendered through the use of the following four basic types. Ambient light provides uniform light to all vertices in the scene; point-lights emit light in all directions; spotlights cast light in a specified cone shape; and, distant lights such as the sun which provide parallel lights. Each light source emits different amounts of light at different frequencies (colours) and can interact with a material in a different manner (Angel, 1997).

### *2.5.2 Materials*

In addition to the type of light and the shading applied, it is useful to know the type of material that the polygon is made of in order to render the surface. There are three basic types of surfaces that are used to simulate different real world materials. The first is a *specular surface*, which makes the surface appear shiny (Angel, 1997). The second type of surface is a *diffuse surface*, which is characterised by the reflected light being scattered in all directions. A *translucent surface* is one surfaces that allows some light to penetrate the surface and to emerge from another location and possibly angle from the object (Angel, 1997).

## *2.6 Geometry Pipeline*

The development of graphics architectures has closely paralleled similar advances in workstations. The key enabling technology in both cases was the ability to build special-purpose VLSI circuits. The other development was the availability of cheap solid-state memory. In addition the development of custom VLSI circuits led to the creation of computer graphics pipeline architectures. These architectures, referred to as geometry pipelines allow the process of rendering 3D graphics to be broken up in to smaller tasks and streamlined (Angel, 1997). The four major steps in the imaging process are:

- Transformation
- Clipping
- Projection
- Rasterization.

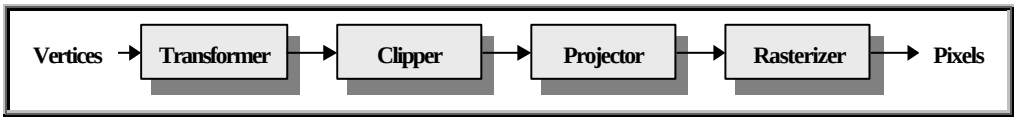


**Figure 2.9: Geometric Pipeline (Angel, 1997)**

### *2.6.1 Transformations*

A number of the steps in the imaging process can be seen as being transformations between the representation of objects in different coordinate systems (Angel, 1997). For example, each object has a local-coordinate system for its vertices. Transformations are applied to locate the object's vertices in the world-coordinate system. Finally, all the vertices in the world-coordinate system are then transformed again to be located in the camera-coordinate system.

### *2.6.2 Clipping*

Clipping is the process of removing objects and vertices that do not appear within the viewable area or *clipping rectangle*. The clipping process can occur at various stages in the imaging process (Angel, 1997). For instance, whole objects are removed if they are completely outside the viewing frustum prior to the application of transformations, while individual vertices in objects that cross the edge of the viewable area are not removed until final projection occurs.

### *2.6.3 Projection*

Eventually the three-dimensional objects need to be projected into two-dimensional objects ready for viewing. The projections used in three-dimensional graphics are known as *planar geometric projections* because the projection is onto a plane rather than a curved surface and uses straight rather than curved projections. There are a number of projection methods available which are used to gain different effects, such as parallel projections and the more commonly used perspective projections (Foley, 1996).

### *2.6.4 Rasterization*

Rasterization is the final stage in rendering a scene and is the process of taking the projected objects and representing them as pixels in the framebuffer (Angel, 1997). The rasterizer uses the projected x, y coordinates as the screen position for a given pixel and the z-coordinate for depth comparison (z-buffering) and the removal of hidden surfaces. It also may use it for more advance features such as fog effects, performing perspective-correct texture mapping, and for w-buffering (another form of depth buffering) (Microsoft, 1999).

# CHAPTER 3

# OpenGL

Since its inception in the eighties the *Open Graphics Library* (OpenGL), based on IRIS GL by Silicon Graphics has become one of the most popular APIs for most 3D visualisation applications and is widely accepted as the standard. Its popularity stems primarily from its being supported by most workstation suppliers and is available for most platforms through third party vendors. It is also one of the easiest APIs to use, is versatile and provides network transparency (Angel, 1997).

## *3.1 Overview*

OpenGL provides several hundred functions that allow a programmer to specify the objects and operations needed to build high-quality graphical environments in both two and three dimensions. It is a rendering only graphics standard, providing no direct support for other system operations such as handling input devices. This restriction on functionality enables OpenGL to be incorporated into any windowed operating system. The only requirement that OpenGL places on the underlying system is that it provides a framebuffer for the rendering of scenes (Segal, 1999).

OpenGL draws primitives such as points, line segments, polygons and pixel rectangles (or bitmaps) into a framebuffer subject to the currently selected mode or state. For example, all vertices will be drawn with the current colour in the colour-mode, until it is changed. Each mode can be changed independently of others and they all interact to determine what eventually ends up in the frame-buffer. After the modes are set and the primitives specified, operations can then be described by sending *commands* in the form of function calls (Segal, 1999).

The rendering speed of OpenGL is achieved by its using the available hardware on a system whenever possible. It also has incorporated the concept of *display lists* from earlier APIs like PEX and PHIGS that allows a series of operations or commands to be sent to the rendering hardware as a block. OpenGL also provides mechanisms for switching required rendering features on or off, allowing an application to optimise rendering for the particular attributes of the available hardware.

### *3.1.1 Libraries*

The OpenGL basic set of libraries used by a typical application, and the organisation of these libraries, for an X Window system environment is shown in Figure 3.1. All the functions begin with the letters *gl* and are stored in a library usually referred to as the *Graphics Library* (GL). This contains the fundamental low-level functions that create and control the scene. The Graphics Utility Library (GLU) is an additional layer on top of the GL library that contains code for common objects and operations, such as spheres. The GL Utility Toolkit (GLUT) is readily available and is platform specific. It provides the minimum functionality that is expected in all windowing systems. The platform specific libraries are called from the OpenGL libraries so the application does not need to refer to them directly (Angel, 1997).

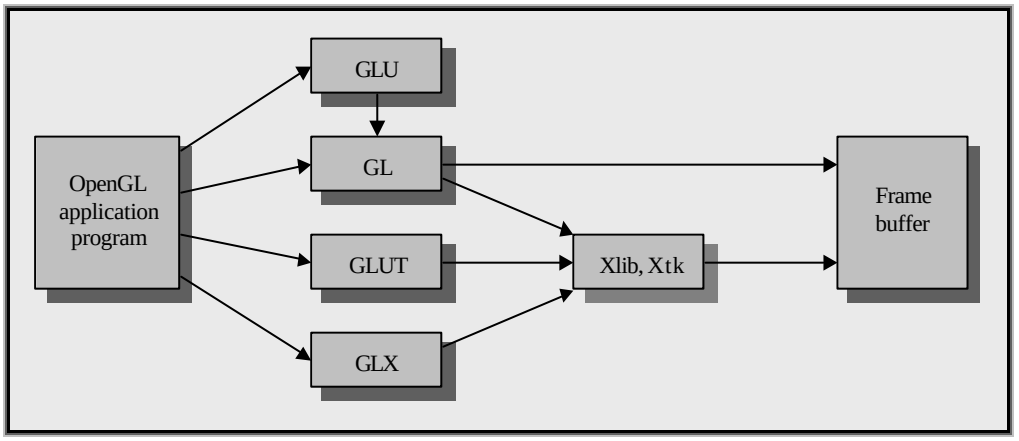


**Figure 3.1: OpenGL Library Organisation for X Window System (Angel, 1997, p45)**

### *3.1.2 Operation*

A schematic diagram of OpenGL can be seen in figure 3.2, showing how commands are received on the left and processed into an image that appears on the frame buffer to the right. Commands can be sent to OpenGL in two ways: either as straight function calls to be acted on immediately or they can be accumulated in the form of a display list ready for processing at a later time.

In the following diagram it can be seen that the pipeline consists of two parts: the top part handles the vertex processing portion of the rendering process, while the lower section handles the 2D pixel rectangle primitives.

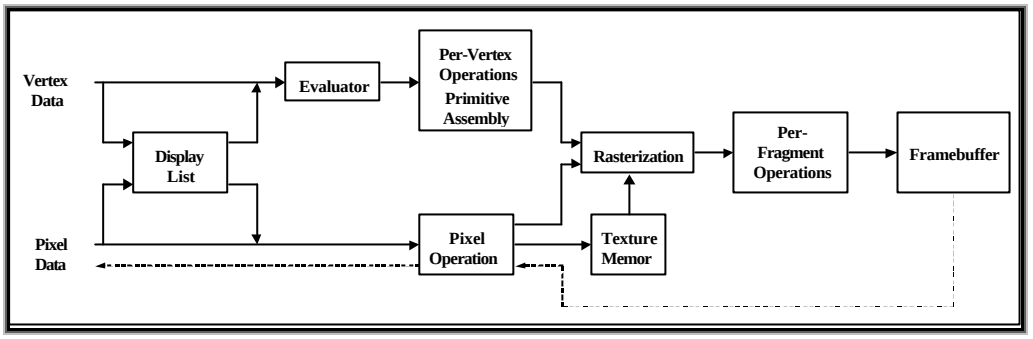


**Figure 3.2: Block Diagram of OpenGL Pipeline (Segal, 1999, p11)**

The first stage of the vertex pipeline, called the evaluator, provides an efficient means for approximating curves such as NURBS (*non-uniform rational B-spline*) or surface geometry through the evaluation of polynomial functions. The second stage operates on the individual vertices of the geometric primitives. The vertices are transformed and lit and the primitives are clipped to the viewing volume. The rasterizer produces a series of framebuffer addresses and values called *fragments.* The last stage performs operations such as depth buffering, blending of incoming fragment colours with stored colours, as well as masking and other logical operations, before updating the framebuffer (Segal, 1999).

The lower part of the diagram shows the stream taken by pixel rectangles or bitmaps. These primitives bypass the vertex-processing portion of the pipeline to either be stored for later use as textures or to send a block of fragments directly through rasterization to the individual fragment operations. Eventually this causes a block of pixels to be written to the framebuffer. You may also notice that values can be read back from the framebuffer or copied form one portion of the framebuffer to another (Segal, 1999).

### *3.1.3 State*

OpenGL uses a client-server model to interpret commands. A program (the client) issues commands, which are received and interpreted by OpenGL (the server). This is done the same way regardless of whether the server is on the same computer as the client or not, making the in between medium transparent. However, if the application is being used over a network and a scene is reasonably static, it is recommended that the program should use display lists in order to achieve best performance.

### *3.1.4 Command Syntax*

In OpenGL, each of the commands that specify vertex coordinates (along with vertex and primitive attributes) comes in a number of different flavours which are distinguished by mnemonic suffixes. This is to accommodate the variations in data formats and numbers of coordinates that applications use. Basically, there are groups of commands that perform the same operation but differ in the number and types of arguments. This makes OpenGL very versatile but also increases its complexity (Segel, 1994).

Commands in OpenGL are made of a name and up to 4 characters. The first is a number, which describes the number of values being presented to the command. The second character or character pair specifies the type of the arguments. Table 3.1 lists the types and the corresponding character code used. The last character is the letter 'v' and is included if the command takes a pointer to an array (or vector) of values rather than individual arguments (Segel, 1999).

| Character Code | OpenGL Type |
|---|---|
| b | Byte |
| s | Short |
| i | Int |
| f | Float |
| d | Double |
| ub | Unsigned byte |
| us | Unsigned short |
| ui | Unsigned int |

**Table 3.1: OpenGL Command Types (Segel, 1999, p8)**

Below are two example prototypes for the command `glVertex`. The first creates a three-coordinate vertex with floating point precision while the second creates a two-coordinate vertex using integers held in an array.

```
void glVertex3f (float x, float y, float z);
void glVertex2iv(int v[2]);
```

## 3.2 Functions

OpenGL provides an extensive selection of functions for defining primitives and for controlling how a scene is to be rendered. OpenGL uses a number of similar types of calls as earlier APIs to define camera position and transformations but differs significantly with its method of defining vertices and associated data. Also, OpenGL was designed to be platform independent and so does not directly provide functions for interacting with the windowing system.

The major difference between OpenGL and earlier APIs like PEX is that rather than using arrays or structures to define vertex and associated data, OpenGL individually defines data with function calls. The advantage with this approach is that data does not need to be stored by the application in a form that is convenient for the API. Another advantage is that by simply combining calls in the appropriate order, different effects may be achieved, such as all vertices defined after a colour change will also have that colour without them all having to be individually set. The disadvantage with this approach is that function calls may be costly and result in poor performance.

### 3.2.1 Primitive Functions

In OpenGL, primitives are specified by declaring a series of coordinate-sets between the command pairs `glBegin` and `glEnd`. Figure 3.3 illustrates how to specify a triangle with vertices (0,0,0), (0,1,0) and (1,0,1).

```
glBegin(GL_POLYGON);
   glVertex3i(0,0,0);
   glVertex3i(0,1,0);
   glVertex3i(1,0,1);
glEnd();
```

**Figure 3.3: Primitive Specification in OpenGL**

Between these two commands any other code can be included such as code to calculate the next vertices to be specified. The exception to this is that most OpenGL commands that do not specify OpenGL vertices and associated information can not appear here. This restriction allows implementations to run in an optimised mode.

There are ten basic primitives provided in OpenGL, given in Table 3.2. Each vertex in these primitives can be specified with two, three or four coordinates. The forth coordinate, if given, specifies a homogeneous three- dimensional location.

| Type | Description |
|---|---|
| GL_POINTS | Each vertex in the set describes the location of a point. |
| GL_LINES | Each pair of vertices describes a line segment. |
| GL_LINE_STRIP | The first Vertex describes the start point. Each subsequent point describes a line segment joining it to the previous point. |
| GL_LINE_LOOP | The same as GL_LINE_STRIP but an extra line segment is added joining the last vertex with first. |
| GL_POLYGON | The same as GL_LINE_LOOP but can do the same operations as triangles and quadrilaterals such as being filled or shaded. |
| GL_TRIANGLES | Each triad of consecutive vertices describes a triangle. |
| GL_TRIANGLE_STRIP | Each vertex after the first two describes a triangle given by that vertex and the previous two. |
| GL_TRIANGLE_FAN | Each vertex after the first two describes a triangle given by that vertex and the previous vertex and the first vertex. |
| GL_QUADS | Each consecutive group of four vertices describes a quadrilateral. |
| GL_QUAD_STRIP | Each pair of vertices after the first two describe a quadrilateral given that pair and the previous pair. |

**Table 3.2: OpenGL Primitive Types**

### *3.2.2 Attribute Functions*

When specifying primitives between `glBegin` and `glEnd` commands, a number of additional pieces of information can also be included to describe the appearance of the object. An attribute is any additional information that determines how the primitive is to be rendered. Thickness of a point or line and the pattern used to fill a polygon are a couple of attributes but the main ones of concern to us are the specification of vertices *current normal*, *current texture coordinates*, and *colour* (Angel, 1997).

The current normal is a three-dimensional vector that may be specified by sending three coordinates that specify it. The point specified gives one end of the vector and the other end is always the origin. The current normal is used in lighting and shading calculations (it saves having to average the face normals and allows more direct control over shading). Colour values can be specified in

two ways: RGBA (Red, Green, Blue, and Alpha) values when initialised to RGBA mode or by using a single colour index (into a colour palette) when initialised to colour index mode. Texture Coordinates can also be specified, using one, two, three or four texture coordinates, which specify how a texture image maps onto a primitive.

Figure 3.4 shows our triangle specification again but this time with colour and current-normals being set as well.

```
GLfloat colours[3][3] = {{1.0, 0.0, 0.0},
                         {0.0, 1.0, 0.0},
                         {0.0, 0.0, 1.0}}

glBegin(GL_POLYGON);
   glNormal3i(0,1,1);
   glColor3fv(colours[0]);
   glVertex3i(0,0,0);

   glNormal3i(0,1,0);
   glColor3fv(colours[1]);
   glVertex3i(0,1,0);

   glNormal3i(1,1,1);
   glColor3fv(colours[2]);
   glVertex3i(1,0,1);
glEnd();
```

**Figure 3.4: Primitive Specification with attributes in OpenGL**

### *3.2.3 Viewing and Transformation Functions*

OpenGL uses a *camera-based view model* called the *synthetic-camera*. This model looks at creating a computer-generated image as being similar to forming an image using an optical system. The specification of the objects is independent of the specification of the camera. Therefore, the camera is a separate entity that can be moved and manipulated without affecting the environment within which it exists (Sowizral, 2000).

In OpenGL the camera is given a location in the virtual world and a direction that it is facing or looking. Then a number of “lenses” can be selected to control the overall appearance of the final image. When primitives are being specified they are being placed in the model’s coordinate system called the *model-view*. The camera has a separate coordinate system where it is always at the origin and looking down the negative z-axis. Initially, the default model-view matrix is the

identity matrix, therefore the camera coordinate system is identical to the models. To move the camera to a different location a series of transformations are performed to the model view matrix (Angel, 1997).

The actual process of generating an image from the model takes two steps in OpenGL. The first is to form the model view matrix to find the camera position, referred to as *eye* coordinates. Then another matrix, called the *projection matrix*, is applied to yield clip coordinates. In order to change these matrices the current matrix mode must first be set so that OpenGL knows which matrix is being altered. Then any number of transformations can be applied to that matrix (Segel, 1999). Figure 3.5 shows a simple example, concerning the setting up of a camera view. The camera is rotated to look towards the negative x-axis and then translated backward along the positive x-axis, by the distance d. The transformations in OpenGL are specified in reverse order. Therefore, rotation will occur first before translation, even though translation is listed first. This is because matrix multiplication operates from right to left (see section 2.3.4 Transform Concatenation).

```
// Setup model view matrix so that the camera is looking
// at the model from the positive x direction.

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef( 0.0, 0.0, -d );
glRotatef( 90.0, 0.0, 1.0, 0.0 );

// Setup projection matrix to use a Orthoganal-Projection.

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho( -1.0, 1.0, -1.0, 1.0, -1.0, 1.0 );
```

**Figure 3.5: Setting camera position and lens in OpenGL.**

OpenGL also maintains a stack of matrices for all matrix modes. The `GL_MODELVIEW` stack can contain up to 32 matrices, while the others can all hold a maximum of 2. This stack system allows much faster switching between views as the matrices do not need to be calculated in between frames (Segel, 1999).

### *3.2.4 Input and Control Functions*

Apart from the most trivial applications some form of interaction with the user is usually required and they would usually like to see the rendered image as well. OpenGL however does not directly support interactivity, or the creation and manipulation of windows and associated interface components. The reason for this is that it was designed to be as portable as possible. Also, it is expected that the operating system in use already have libraries in place to handle windows and input devices.

The problem with not providing such functionality is that an application would have to rewrite that portion of the program for each platform. Therefore, in the GLUT library comes a set of very basic functions for handling the creation of windows, handling of window events, and accessing input devices. This set of functions has been kept very minimal so that all systems have them. If more functionality is needed then the programmer must use operating specific calls (Angel, 1997).

A basic template for an OpenGL application, using GLUT, is shown in Figure 3.6. It uses a single window for rendering and a mouse to receive input. Firstly, before a window can be opened, there must be interaction between GLUT and the windowing system, which is initiated with the call to `glutInit`. Then a window is created by first setting the display mode: in this case it is using RGB rather than indexed colour; a depth buffer for hidden surface removal; and double rather than single buffering (for page flipping). The window is also given a size and starting location on the screen. Finally, the call to create the window also gives it a name. Then the mouse is set up to get input by passing a pointer to a callback function, to `glutMouseFunc`, which is called whenever a mouse event occurs. That function is defined to perform the required operations. The `myInit` function is used for any initialisation code and `glutMainLoop` is a never-ending loop in GLUT that waits for events from the operating system. While GLUT may not provide a lot of functionality, this example shows it is not particularly difficult to use.

```
#include <GL.h>
#include <glut.h>

void main(int argc, char **argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
   glutInitWindowSize(200, 100);
   glutInitWindowPosition(50, 50);
   glutCreateWindow("Simple OpenGL example");
   glutDisplayFunc(display);
   glutMouseFunc(mouse);
   myInit();
   glutMainLoop();
}
void myInit( void )
{
   // Initialisation code goes here.
}
void display( void )
{
   // Screen rendering code goes here.
}

void mouse(int btn, int state, int x, int y)
{
   if(btn==GLUT_LEFT_BUTTON&&state++GLUT_DOWN)
      exit();
}
```

**Figure 3.6: A Basic Application Template for OpenGL using GLUT.**

## *3.3 Conclusion*

The OpenGL graphics API was intended for use in interactive applications. It was designed to provide access to the graphics hardware capabilities regardless of the level of those capabilities. It is supported on all the major windowing systems making applications that use the API more portable. Its flexible interface means that no particular method of describing objects is enforced on applications. This minimalist structure provides an excellent base for libraries to be built for handling structured geometric objects, regardless of the structure used. The result is a straightforward API with few special cases, which is easy to use in a variety of applications requiring high performance on many different platforms.

# CHAPTER 4

# Direct3D

Direct3D is a member of the DirectX family of APIs and is designed to enable world-class game and interactive graphics development on the Windows platform. It provides device-independent access to 3D video-display hardware with a common interface. It is tightly integrated with the other APIs in the DirectX collection, providing a highly versatile environment for game development. DirectX has become the leading API for game development on the IBM PC (Microsoft, 1999).

## *4.1 Overview*

When Microsoft released the Windows 95 operating system they also decided to try and corner the game development industry by developing a set of APIs under the DirectX umbrella. DirectX offers portable access to the features used in MS-DOS and removes the obstacles to hardware innovation on the PC. It provides a consistent interface between hardware and applications, thereby reducing the complexity of installation and configuration as well as utilising the hardware to its best advantage (Microsoft, 1995).

DirectX is highly versatile, providing a number of methods for performing most operations. It also provides direct control over all features available on the system in a standard interface. Its speed is generated by its ability to utilise the available hardware on the system and should the hardware not be available it uses efficient algorithms in software to emulate the hardware. DirectX provides simple data structures for describing vertex information that can be sent to the renderer in streams thereby limiting the number of function calls (Microsoft, 1999).

One of the greatest advantages with DirectX though, is that it is not a standard and so can be regularly updated, to keep up with the developing hardware technology, without having to go through extensive review procedures. Therefore, developers are constantly able to utilise the most advanced features from within the same API. Conversely, OpenGL has remained virtually stagnant leaving developers unable to access the new technologies (anon (a), 1999).

### *4.1.1 Libraries*

DirectX is a large collection of APIs that can be interconnected to provide full and complete control of the entire computer, allowing an application to utilise all the power of the underlying hardware. This paper is primarily concerned with the Direct3D Immediate Mode API but to provide a general overview the table below lists and briefly describes each of the APIs available with DirectX.

| API Library | Description |
|---|---|
| DirectDraw | DirectDraw allows an application to directly manipulate the display memory, the hardware *blitter*, hardware overlay support, and flipping surface support. It accomplishes this while maintaining compatibility with the Windows Graphic Device interface (GDI). (Microsoft, 1999). |
| Direct3D Immediate Mode | Is a low-level API that provides a device-independent access to 3D hardware and is ideal for developers of high-performance multimedia based applications (Microsoft, 1999). |
| Direct3D Retained Mode | In the latest release, DirectX7, this API has been replaced by Direct3DX and is not being updated any further (but is still included for backward compatibility). Essentially Retained Mode provided a higher level API (Microsoft, 1999). |
| Direct3DX Utility Library | The Direct3DX library is a helper layer that sits on top of the immediate mode API. It simplifies or eliminates difficult or tedious tasks (Microsoft, 1999). This library is the first step in the future incorporation of the, currently still being developed Fahrenheit API in a later version of DirectX (Torborg, 2000). |
| DirectInput | DirectInput is an API for input devices such as the mouse, keyboard, joysticks and other input controllers as well as for input/output force-feedback devices. It provides a method of directly accessing these devices rather than relying on windows messages (Microsoft, 1999). |
| DirectSound | DirectSound provides low-latency mixing, hardware acceleration, and direct access to the sound device and enables wave sound capture and playback (Microsoft, 1999). |
| DirectMusic | DirectMusic works with message based musical data which are converted to wave samples which are then streamed to DirectSound (Microsoft, 1999). |
| DirectPlay | DirectPlay provides network capabilities for games, game servers and other applications without writing code for individual protocols (Microsoft, 1999). |
| DirectSetup | DirectSound provides a single call installation of the DirectX components (Microsoft, 1999). |

**Table 4.1: DirectX Components.**

### *4.1.2 Architecture*

DirectDraw and Direct3D provide device independence through the *Hardware Abstraction Layer* (HAL). The HAL is a device specific interface provided by the manufacturer of the device. The respective APIs act as the intermediaries so that the application programmer never actually has to interact directly with the hardware. However, all hardware devices provide different levels of functionality, therefore a second layer above the HAL is provided to emulate the function, called the *Hardware Emulation Layer* (HEL). The result is transparent support for all major features regardless of hardware (Microsoft, 1999).

Figure 4.1 below shows the relationship between the DirectDraw, Direct3D and Direct3DX interfaces and the *Graphics Device Interface* (GDI) (the standard Windows graphics API), the HAL, the HEL and the hardware. Direct3D is tightly integrated with the DirectDraw component of DirectX. DirectDraw surfaces are used as rendering targets and as z-buffers. Therefore, the Direct3D interface is actually an interface to a DirectDraw object (Microsoft, 1999).

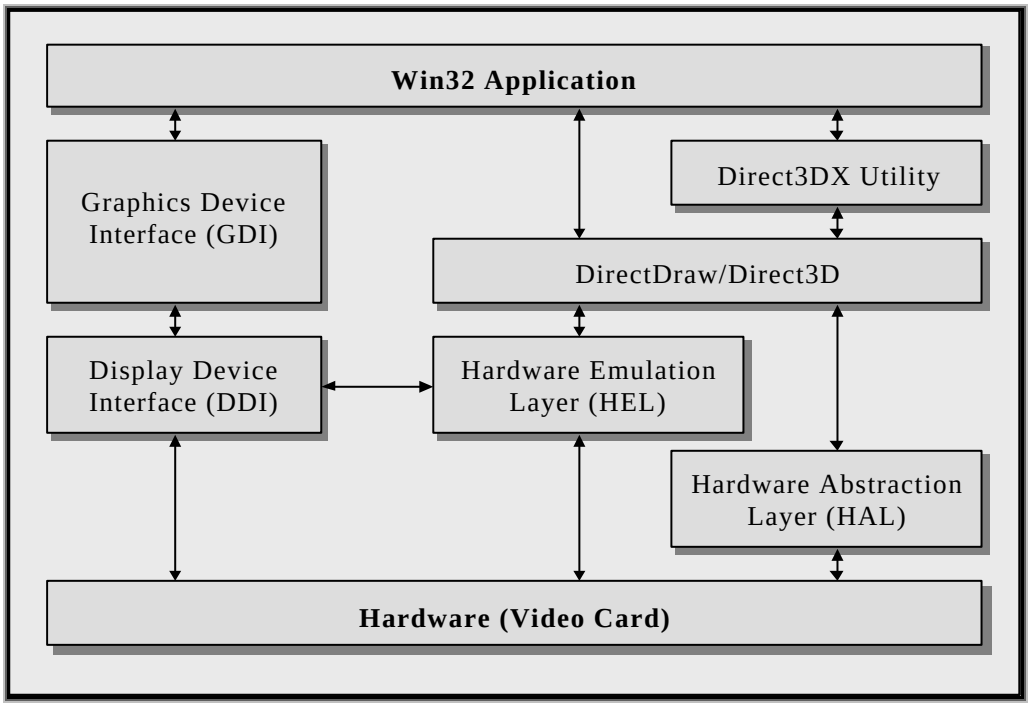


**Figure 4.1: DirectX Integration (Microsoft, 1999)**

### *4.1.3 Object Types*

DirectX is an object-oriented API implemented using the *Component Object Model* (COM). COM is the foundation of an object-based system that focuses on interface reuse. Direct3D is made up of a series of objects and the developer works with these objects to manipulate their virtual world (Microsoft, 1999).

The **DirectDraw** object provides the functionality of Direct3D. It is the heart of any Direct3D application and is the first object created and all other related objects are made from it. Direct3D is incorporated into DirectDraw because DirectDraw represents the display device, which implements many of the most important features of Direct3D (Microsoft, 1999).

The **DirectDrawSurface** object, casually referred to as a '*surface*', represents a block of memory used to store data to be displayed. It can be in either system memory or video memory (video memory is limited but much faster). It can be used in flipping chains, to store bitmaps and textures, and is the target for a Direct3DDevice when rendering (Microsoft, 1999).

The **Direct3DDevice** object encapsulates and stores the rendering state of the application. It performs transformations and lighting operations and finally rasterizes an image to a DirectDraw surface. DirectX also allows applications that use custom transformation and lighting models to bypass these stages of the rendering pipeline. Direct3DDevice objects are independent of the surface they are rendering to and can render to multiple surfaces (Microsoft, 1999).

The **Direct3DVertexBuffer** object is a memory buffer that contains vertices to be rendered with the vertex-rendering methods available in the Direct3DDevice object. Direct3DVertexBuffers provide a means for reusing already transformed geometry. After the vertices have been transformed, lit and clipped it can be rendered repeatedly even with interleaved render state changes such as multiple textures (Microsoft, 1999). Vertex buffers can also be optimised to organise the vertices in a format that is best suited for the particular hardware pipeline. Mark Kenworthy, Group Program Manager for DirectX at Microsoft, claims that there is "…about a 30% performance boost from using vertex buffers" (Kenworthy, 2000).

## 4.2 Functions

Many of the DirectX header files include macro definitions for each method. These macros serve two purposes: they can simplify the use of the methods and can expand into appropriate calls in C or C++ syntax, thereby, providing a non-object-oriented interface for C programmers. DirectX uses similar methods to OpenGL and other APIs to define camera positions and transformations but differs significantly to OpenGL when defining vertices and associated data. While DirectX has the disadvantage of only being available for the IBM-PC it does provide extensive methods for interacting with all aspects of the computer system.

### 4.2.1 Primitive and Attribute Functions

Unlike OpenGL, there are no special functions for creating primitives and they can be created anywhere at any time prior to rendering. Primitives in DirectX consist of an array of vertices that are identified as being a particular type of primitive when they are sent to the renderer. Therefore, the same array of vertices can be rendered as a mesh of shaded triangles, a wire frame or a set of individual points, by simply changing a flag in the rendering method call. DirectX primitive types are a subset of the OpenGL types and the flags for these types are D3DPT_POINTLIST, D3DPT_LINELIST, D3DPT_LINESTRIP, D3DPT_ TRIANGLELIST, D3DPT_TRIANGLESTRIP, and D3DPT_TRIANGLEFAN. For a description of these types see the similarly named types in Table 3.2.

The vertices that make up primitives can be in many different forms as well, provided that a single primitive only consists of a single type of vertex. This use of what DirectX calls *Flexible Vertex Formats* (FVF) is very powerful, in that an application only needs to define the vertex components that it requires. While this can complicate DirectX code it has the advantage of conserving memory and reducing rendering time (Microsoft, 1999).

The various rendering methods in Direct3D take a combination of flags, so that the Direct3DDevice can determine how to render the primitive. These flags inform the system which vertex components the application uses and, indirectly,

which parts of the rendering pipeline should be applied to them (Microsoft, 1999). Figure 4.2 shows the various components that an application's vertices can be composed of and the required order the data must be presented in.

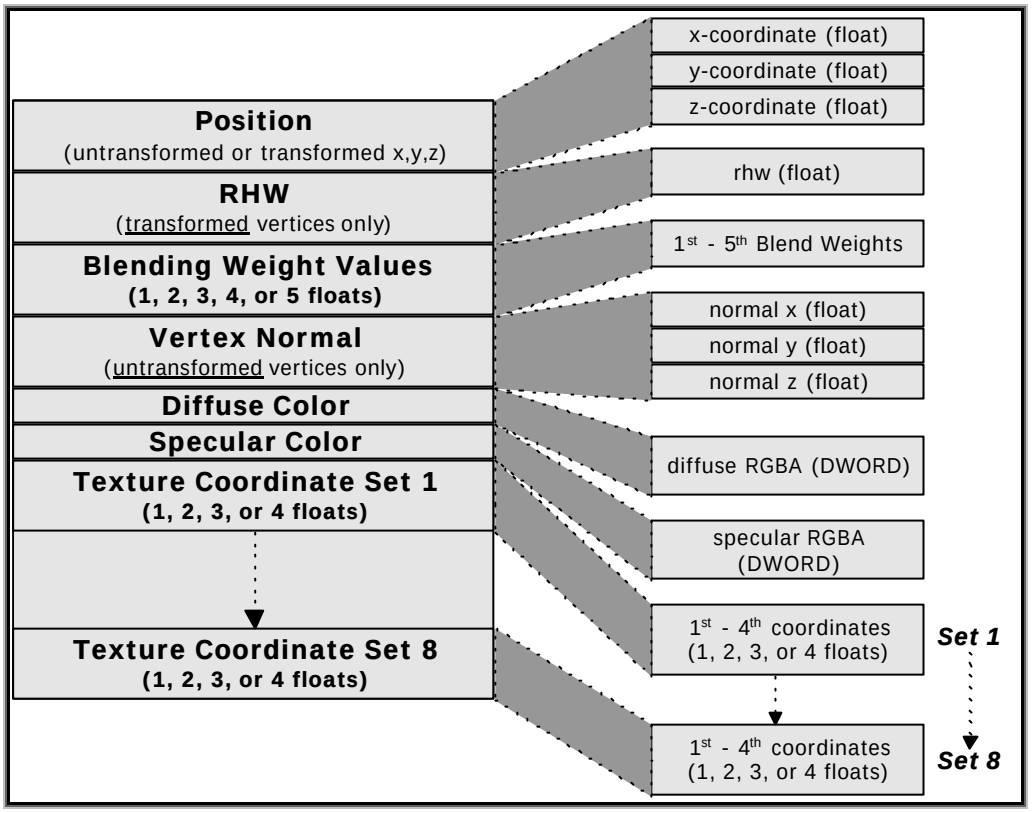


**Figure 4.2: DirectX Vertex Data Format (Microsoft, 1999, p181).**

It may be noticed that textures can have various amounts of coordinates. This is because they can be declared in different formats. Also, not all of the above components can be used together. For instance, it makes no sense to define a vertex with a normal and a RHW value as the former is for untransformed vertices while the latter is for transformed.

Colour is not directly attached to particular vertices, as it is in OpenGL. Rather, the rendering method calculates colour by using the currently set material and lighting sources. Materials are structures that define the way different types of light are reflected by the primitive (see 2.5.2 Materials) (Microsoft, 1999).

Figure 4.3 shows a definition of a single, one sided, triangle (similar to the one defined using OpenGL in Figure 3.4) setting the current material and an ambient light source.

```
D3DVERTEX           g_pvTriangleVertices[3];

HRESULT InitPrimitives(LPDIRECT3DDEVICE7 pd3dDevice)
{
   // Define vertex coordinates
   D3DVECTOR p1( 0.0f, 0.0f, 0.0f );
   D3DVECTOR p2( 0.0f, 1.0f, 0.0f );
   D3DVECTOR p3( 1.0f, 0.0f, 1.0f );

   // Define Normal Vectors
   D3DVECTOR n1( 0.0f, 1.0f, 1.0f );
   D3DVECTOR n2( 0.0f, 1.0f, 0.0f );
   D3DVECTOR n3( 1.0f, 1.0f, 1.0f );

   // Construct vertex structures
   g_pvTriangleVertices[0] = D3DVERTEX( p1, n1, 0, 0 );
   g_pvTriangleVertices[1] = D3DVERTEX( p2, n2, 0, 0 );
   g_pvTriangleVertices[2] = D3DVERTEX( p3, n3, 0, 0 );

   // Create material used by triangle to reflect only red
   // and blue light and set the 3D device to that material.
   D3DMATERIAL7 mtrl;
   ZeroMemory(&mtrl, sizeof(mtrl));
   mtrl.ambient.r = 1.0f;
   mtrl.ambient.g = 1.0f;
   mtrl.ambient.b = 0.0f;
   pd3dDevice->SetMaterial(&mtrl);

   // Set up an ambient light source.
   pd3dDevice->SetRenderState(D3DRENDERSTATE_AMBIENT,
                              0xffffffff);
   return S_OK;
}
```

**Figure 4.3: Primitive Specification with attributes in DirectX.**

### *4.2.2 Viewing and Transformation Functions*

DirectX, like OpenGL, also uses the camera-based view model and in a lot of ways is very similar to use. The camera is given a location by setting up a *view matrix*, transforms the vertices from the world coordinate system to the camera's coordinate system. Due to DirectX using a left handed coordinate system the camera is, initially, looking down the *positive* z-axis. Secondly, the application must set up a *projection matrix*, which sets the front and back clipping planes as well as the horizontal and vertical field of view angles (Microsoft, 1999). Figure 4.4 illustrates how to create both a view and projection matrix.

```
HRESULT InitMatrices(LPDIRECT3DDEVICE7 pd3dDevice)
{
   // Initiallise matrices to identity matrix.
   D3DMATRIX matView, matProj;
   D3DUtil_SetIdentityMatrix(matView);
   D3DUtil_SetIdentityMatrix(matProj);

   // Move Camera backwards along the negative z-axis.
   matView._43 = 10.0f;

   // Set the view matrix to be used by the device.
   pd3dDevice->SetTransform(D3DTRANSFORMSTATE_VIEW,
                            &matView);

   // Create standard perspective projection matrix.
   matProj._11 =  2.0f;
   matProj._22 =  2.0f;
   matProj._34 =  1.0f;
   matProj._43 = -1.0f;
   matProj._44 =  0.0f;

   // Set the projection matrix to be used by the device.
   pd3dDevice->SetTransform(D3DTRANSFORMSTATE_PROJECTION,
                            &matProj);

   return S_OK;
}
```

**Figure 4.4: Setting View and Projection matrices in DirectX (Microsoft, 1999).**

### *4.2.3 Input and Control Functions*

DirectX is a fully integrated system and offers full support to all aspects of the computer system. This includes full control over video hardware along with input devices, sound hardware and network connections. This gives the developer significant benefits in controlling and fine tuning their applications over an API like OpenGL, which only offers minimum support in this area (Microsoft, 1999). Often OpenGL applications use DirectX for window, sound and input device management rather than GLUT when run on IBM-PCs.

It may have been apparent to the reader, though; that some of the code examples of DirectX are more complicated than in OpenGL. The complexity of DirectX is made even more apparent when initialising and managing windows and input devices. One of the biggest complaints against DirectX has been the complexity in simply setting up the system. For instance, initialisation of DirectDraw and Direct3D would take approximately 80 lines of code.

This has been rectified in the latest release by the introduction of what is called a *Context*, and is found in the Direct3DX library. A Context is a COM object that encapsulates all of the Direct3D and DirectDraw objects that are needed for rendering. While you can still use the immediate mode initialisation for finer control of the set up, using the Context object reduces initialisation to just a single function call to `D3DXCreateContext` or `D3DXCreateContextEx` (which offers access to more advanced features)(Microsoft, 1999).

## *4.3 Conclusion*

The DirectX set of APIs offers portable access and a consistent interface between hardware and applications. It is a highly versatile low-level API and gains its speed by the developers being able to fine-tune every aspect of the system and prevent it from having to do unnecessary work. It has also been regularly providing new features pushing both hardware and software developments to greater stages of advancement.

For these reasons it has become the market leader in game development and threatens OpenGL's dominance of the CAD and CAE environments. There are three main problems with DirectX. Firstly, it is an extremely complicated API, therefore, simple tasks demand significant development and projects require a high level of maintenance. Secondly, as a result of constant updating developers need to keep learning each new release. In addition it is only available for the IBM-PC.

# CHAPTER 5
# Java 3D

Java 3D is a new and burgeoning fourth generation API that can be used for writing stand-alone 3D graphics or Web-based 3D applets. Its specification is a result of a joint collaboration between Silicon Graphics, Inc., Intel Corporation, Apple Computer, Inc., and Sun Microsystems, Inc. Java 3D provides a development environment that abstracts away from the underlying complexities found in low-level APIs, making 3D programming more accessible to new developers (Sowizral, 2000).

## *5.1 Overview*

Java 3D represents an evolution to a high-level 3D API that offers a high degree of interactivity while preserving platform independence. Java 3D is a retained mode API (see 5.1.4 Rendering Modes) that uses a technology known as *Scene Graphs* based on techniques used in PHIGS and SGI. The goals of the developers of Java 3D were to produce a high performance API that offered a rich set of 3D features for creating interesting 3D worlds. This was to be achieved using a high-level object-oriented paradigm that could support a wide variety of file formats such as vendor-specific CAD formats, interchange formats and VRML (Sowizral, 2000).

The greatest difficulty in developing Java 3D was accessing the underlying video hardware to maximise performance. To do this Java 3D has been layered using OpenGL and Direct3D as its low-level rendering APIs. Other optimisations have also been included such as *state sorted rendering* which orders objects according to their characteristics, and the ability to *compile* a scene graph into an optimised format for the underlying rendering API. Java 3D also implements a *view frustum culling* that removes objects outside the viewing area (anon (b), 1999).

Java 3D was implemented to help developers with little or no graphics programming experience use 3D graphics in their applications (Day, 1998). It does this by eliminating many bookkeeping chores of low-level APIs. It allows the programmer to think about geometric objects rather than triangles - about a scene and its composition rather than how to write efficient rendering code (Sowizral, 1997).

### *5.1.1 Scene Graph Basics*

A scene graph in Java 3D is a hierarchical tree structure containing a number of nodes. The user can create one or more subgraphs and attach them to a *virtual universe*. The connections between nodes always represent a direct parent to child relationship. Scene graphs in Java 3D can not contain any cycles, therefore, they are a directed acyclic graph (DAG) (Sowizral, 1997).

The DAG approach to representing a scene allows the same data, such as geometry and attributes, to be used multiple times. This saves memory, start-up time and reduces download time over a network making this technique particular valuable to the Java environment. This is especially useful when a scene contains large quantities of redundant information. The disadvantage with this approach is that rendering can be slower and more difficult because of the need to traverse processes due to the dynamically bound objects and the limited navigation within the DAG (Beier, 1995).

The nodes in a Java 3D scene-graph can be either a *Group* node or a *Leaf* node. Group nodes are the glue elements used in constructing a scene graph. There are seven types of Group node, each of which has special roles. They all can have zero or more child nodes and a maximum of one parent. The only exception is the SharedGroup node, which has no parents. A Leaf node contains the actual definitions of shapes (geometry), lights, fog, sounds and so forth. A leaf node can only have one parent and no children (Sowizral, 2000).

The state and graphics context of a leaf node is defined by a linear path between the root and that node. This allows the Java 3D renderer to traverse the scene graph in whatever order it wishes. This significantly improves rendering speed as the renderer can select the best method for the particular hardware it is running on and negates the earlier mentioned disadvantage of using DAGs. This is also in marked contrast to earlier APIs such as PHIGS and SGI that used a scene graph-based structure called display lists, where if a node changes its graphics state then all nodes to the right and below it were effected (Sowizral, 2000).

### *5.1.2 Scene Graph Superstructure*

Once a scene graph object has been created it is connected to other scene graph objects in the form of a subgraph. A subgraph can be attached to other subgraphs using a BranchGroup node or can be attached to a VirtualUniverse via a high-resolution Locale object. These objects are scene graph superstructures and are used to contain subgraphs (Sowizral, 2000).

A VirtualUniverse is a node object that consists of a list of Locale objects which in turn contain a collection of scene graph nodes that exist in the universe. Typically, an application will only need one VirtualUniverse, even for very large virtual databases (Sowizral, 2000).

The Locale object is a container of subgraphs of the scene graph that are rooted by a BranchGroup node. It also defines a position in the virtual universe using high-resolution coordinates (HiResCoord). This position acts as the origin for all scene graph objects contained in the Locales subgraph. High-resolution coordinates use three 256-bit fixed-point numbers, one each for x, y, and z (Sowizral, 2000).

This system is sufficient to describe a virtual universe of several hundred billion light years across, while still being able to define objects smaller than a proton. HiResCoord coordinates are only used to embed more traditional floating-point coordinate systems. This method allows a visually seamless virtual universe of any conceivable size without worrying about numerical accuracy (Sowizral, 2000).

### *5.1.3 Scene Graph Example*

Figure 5.1 illustrates a scene graph for a simple application. The example scene graph consists of two superstructure components: a VirtualUniverse and a single Locale. There are then two subgraphs rooted by BranchGroup nodes attached to the Locale. The left subgraph contains two subtrees and is called the *content branch* (Day, 1998). The first subtree is a user extended behaviour leaf node that controls and manipulates the transform matrix associated with the object's geometry. The second subtree consists of a TransformGroup node and a single child, a Shape3D node, which refers to two node component objects: Geometry and Appearance.

The right subgraph, called the *view branch*, contains a single subtree that consists of a TransformGroup node, which specifies the position, orientation, and the scale of the ViewPlatform leaf node. The ViewPlatform object defines the end user's view of the VirtualUniverse. With many of its parameters coming from the View object and its associated objects (Sowizral, 1997).

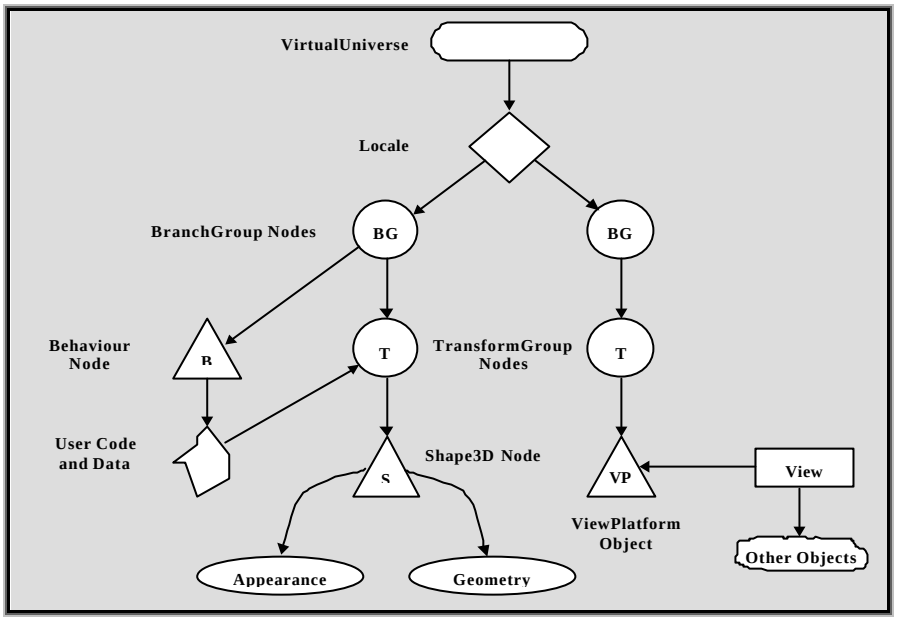


**Figure 5.1: Java 3D Scene Graph Example (Sowizral, 1997, p9).**

### *5.1.4 Rendering Modes*

Java 3D provides three rendering modes: *immediate mode*, *retained mode* (the default mode), and *compiled-retained mode*. Each successive mode provides Java 3D greater freedom in optimising a particular application's execution (Sowizral, 1997).

**Immediate mode** provides very little optimisation. This mode provides more freedom to the developer at the expense of rendering speed. The application does not have to define a scene using scene graphs but does have to provide a draw method with a complete set of points, lines, or triangles (Sowizral, 2000).

**Retained mode** requires an application to build a scene graph and specify which elements of the graph may change during rendering. The use of this rendering method allows Java 3D to improve the rendering speed markedly (Sowizral, 1997).

**Compiled-Retained mode** is an extension of retained mode where the developer can call the `compile` function on all or some subgraphs. Java 3D then optimises the data for the system that its being run (Sowizral, 1997).

## 5.2 Functions

Java3D provides a vast library of classes for building and manipulating large virtual worlds. Most of these classes only expose accessor and mutator methods that operate only on a particular object's internal state. While applications can extend Java 3D classes and add their own methods in the usual way, they can not override Java 3D's scene graph traversal semantics. This is due to the fact that the nodes do not contain explicit traversal and draw methods as the Java 3D renderer retains those semantics internally. However, Java 3D does provide hooks for mixing Java 3D scene graph rendering with user-defined rendering using Java 3D's immediate mode constructs (Sowizral, 2000).

### 5.2.1 Primitive and Attribute Functions

In Java 3D primitives and their attributes are specified through the creation of leaf nodes. Leaf nodes specify all the elements of a scene and its behaviour such as geometry, sounds, lights as well as special linking and instancing capabilities for sharing scene graphs. Table 5.1 lists and briefly describes a small selection of available leaf node objects (Sowizral, 1997).

| Node | Description |
|---|---|
| Background Node | Defines a solid background colour or a background image to fill the window at the beginning of every new frame. |
| Behavior Node | Allows an application to manipulate a scene graph during run time. |
| Fog Node | Specifies the attributes that control fog or depth queuing in the scene. |
| Light Node | Abstract class of which there are a number of defined subclasses that define light sources. |
| Link Node | Allows an application to reference a SharedGroup node from within a branch of the scene graph. Numerous Link nodes can reference the same SharedGroup node making one definition appear numerous times. |
| Shape3D Node | Specifies all geometric objects and contains two references. The first is for the shape's geometry and the second is for the shape's appearance. |
| Sound Node | Abstract class of which there are a number of defined subclasses that define sound sources. |
| ViewPlatform Node | Defines a viewing platform that is referenced by a View object. |

**Table 5.1: Selection of Java 3D leaf node objects (Sowizral, 1997).**

In many of these leaf node objects a reference to more complex entities, called node component objects, is utilised. These node component objects encapsulate related state information in a single entity. There are two main groups of these in Java 3D: geometry component objects and attribute component objects (Sowizral, 1997).

Geometry component objects describe both the geometry and topology of the Shape3D nodes, which reference them. There are four generic geometry types that describe a visible object or set of objects. The fourth type; GeometryArray's, has a number of subclasses that specify arrays of vertex components: coordinates, colours, normals and textures. It also has a bitmask that indicates which components are present (Sowizral, 1997). The type of GeometryArray objects available closely mirrors the primitive types in OpenGL (see Table 3.2).

Attribute component objects provide additional information about an entity such as materials, textures and images or alternatively they can implement other properties like bounding boxes to the nodes that reference them (Sowizral, 1997). Figure 5.2 gives a small program fragment that illustrates how to define a triangle (Sowizral, 2000).

```
final int NumVertices = 3;
float vertices[][]={{0.0f, 0.0f, 0.0f},
                    {0.0f, 1.0f, 0.0f},
                    {1.0f, 0.0f, 1.0f}};

float colours[][] ={{1.0f, 0.0f, 0.0f},
                    {0.0f, 1.0f, 0.0f},
                    {0.0f, 0.0f, 1.0f}};

float normals[][] ={{0.0f, 1.0f, 1.0f},
                    {0.0f, 1.0f, 0.0f},
                    {1.0f, 1.0f, 1.0f}};

// Create an empty triangle.
TriangleArray myTriangle = new TriangleArray
               (NumVertices, COORDINATES|NORMALS|COLOR_3);

// Fill the triangle with data.
for (int i=0;i<3;i++){
     myTriangle.setCoordinate(i, vertices[i]);
     myTriangle.setColor(i, colours[i]);
     myTriangle.setNormal(i, normals[i]);
}
```

**Figure 5.2: Primitive Specification with attributes in Java 3D**

### *5.2.2 Viewing Functions*

Java 3D has introduced a new view model, which takes advantage of the Java philosophy of “write once, run anywhere”. The new system allows Java 3D to generalise viewing operations so that a broad range of display devices can be used such as flat screen displays, portals/caves, and head-mounted displays, without having to modify any code (Sowizral, 1997).

It has achieved this by cleanly separating the virtual world from the physical world. Simultaneously, it also builds a bridge between these two worlds by defining a mapping between the virtual space surrounding the ViewPlatform and the physical space that is specified by the View object. Therefore, a point in physical space corresponds to a point in virtual space (Sowizral, 1999).

Both OpenGL and DirectX utilise a camera-based view model that emulates a camera in the virtual world. In these systems the developer must continually keep re-positioning the camera to emulate a human in the virtual world. While these systems allow the programmer full control over all rendering parameters, they can also be problematical in systems where the physical environment dictates some of the view parameters. For example, the optics of a *head-mounted display* (HMD) is directly determined by the field-of-view that the application should use. As a result of the different optics of each HMD allowing end users to vary these parameters is a poor solution (Sowiral, 1997).

The process of building a view in Java 3D involves including a number of different objects in a specific way as illustrated in Figure 5.3. Firstly, a ViewPlatform node is added to the scene graph that defines a coordinate system. It acts as a point of attachment for View objects and is a base for determining a renderer’s view. Its parents determine the ViewPlatforms location and orientation within virtual space. An application can alter the TransformGroup node parent to move the view around virtual space. Java 3D also provides the facility of having multiple ViewPlatforms. By detaching the View objects from the ViewPlatform and attaching it to another the screen can instantly show a different view of the scene (Sowizral, 2000).

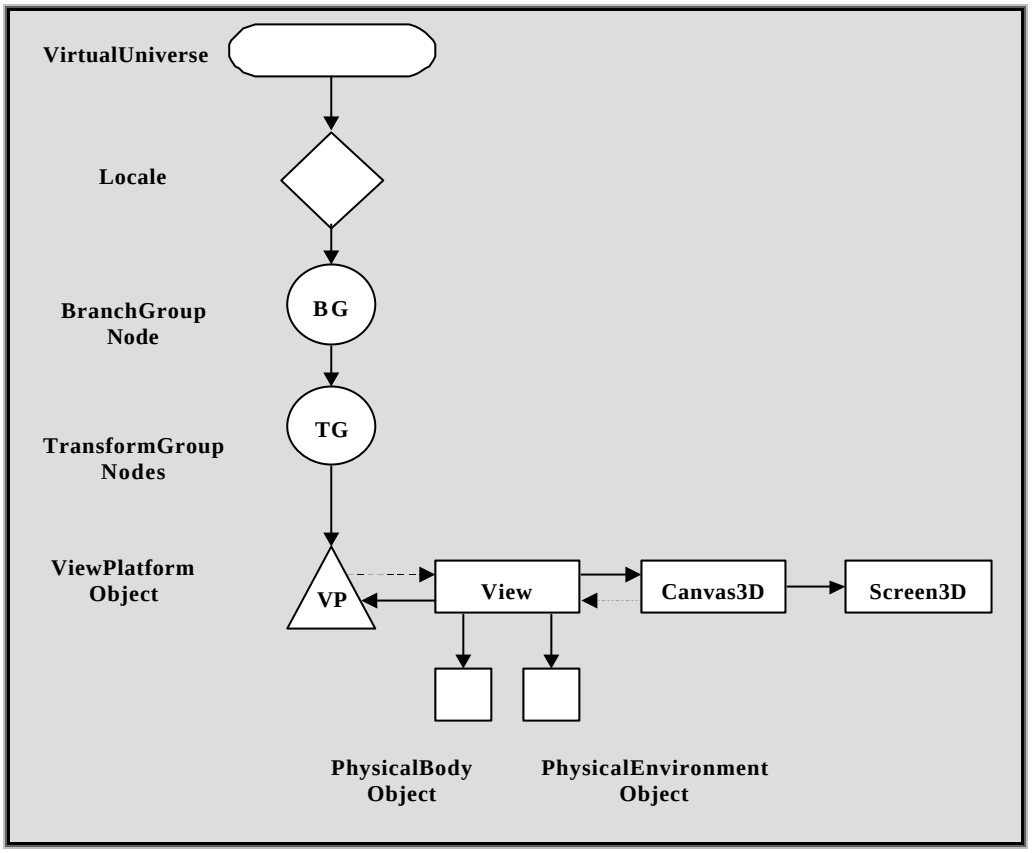


**Figure 5.3: Set up of View objects in Java 3D (Sowizral, 2000).**

The ViewPlatform is then attached to a View object, which specifies the required information for rendering the scene. The View object is the central object for coordinating all aspects of viewing. It requires a Canvas3D object (similar to the Canvas object in the standard AWT) for rendering into. The Canvas3D also requires a Screen3D which encapsulates all of the parameters associated with the physical screen. Optionally, a View object can also have two other objects associated with it. The first is called the PhysicalBody, which contains all the parameters associated with the end-users physical body characteristics such as head position, as well as right and left eye position. The second is the PhysicalEnvironment object, which encapsulates all the parameters associated with the physical environment.

### *5.2.3 Transformation Functions*

Transformations in Java 3D are applied through the inclusion of TransformGroup nodes in a scene graph. A TransformGroup node specifies a single transformation via a Transform3D object and can position, orient, and scale all of its children. Also, the effects of these nodes are cumulative; therefore, if there is more than one TransformGroup node that is in a path between a leaf node and the Locale then the transforms are concatenated into a *composite model transform* (CMT) (Sowizral, 2000). Essentially, the Transform3D class is a simple *wrapper* to separate the developer from the lower level matrix manipulation of the other third generation APIs.

### *5.2.4 Input and Control Functions*

Java 3D provides a number of features such as networking, multitasking and support for basic input devices such as keyboards and mice through the libraries that are already included in the standard Java. However, Java 3D also supports a number of continuous input devices like six-degrees-of-freedom (6DOF) trackers and joysticks (Sowizral, 2000).

For these time-critical devices, Java 3D introduced a new input model. Java 3D has abstracted away the notion of a tracking device or joystick in the form of a sensor. These sensors consist of a time-stamped sequence of values and the state of buttons and switches at the moment of sampling. Typically, though, a particular hardware device may have a number of sensing elements. In these situations a number of sensors may be set up into an array. Developers can access a sensor's data either directly through Java code or they can assign the sensor to one of Java 3D's predefined 6DOF entities such as UserHead (Sowizral, 2000).

To process events generated by a mouse, keyboard or sensor, Java 3D has provided a *Behavior* node. A *Behavior* node is added by attaching it to awake when a particular event occurs such as a mouse click. Once activated, a *Behavior* node can interact with other objects by changing node values within a scene graph such as changing a transform to rotate a particular shape. *Behavior* nodes can also be called from a process schedular, thereby allowing it to perform animations (Sowizral, 2000).

## *5.3 Conclusion*

Java 3D provides a fully integrated virtual reality, platform independent and hardware independent, development environment that is especially powerful when applied to networked applications. Its scene graph-based technique provides a system of sending data over a network only once. It also has the added facility of being able to compress geometry providing Java 3D with a strong edge. In addition, it provides a high-level, object-oriented interface for developers. Java 3D, therefore, gives new developers to 3D programming the opportunity to create cutting edge graphics with far less training than in previous APIs.

However, Java 3D's abstraction from the implementation details also limits some of the API's versatility. For example, it is more difficult to switch on and off particular rendering features and extending its capabilities, while possible, is significantly more difficult and usually means that Java 3D's features are lost in the process. Also, being built on the Java platform and due to the use of scene graphs it is slower than the immediate mode APIs. Therefore, while Sun's literature claims Java 3D can be used for Web-based CAD systems and 3D-games, anecdotal evidence would suggest that it is unlikely to be used frequently for such intense interactive environments by developers in the near future.

# CHAPTER 6

# *Conclusion*

This paper has looked at the two most popular and widely used APIs, OpenGL and DirectX, and a third that has only recently entered the market, Java 3D, with the purpose of comparing their programability, versatility, and innovation in furthering the development of interactive 3D graphics and virtual reality environments. Initially, this paper looked through history at the earlier APIs to investigate various developments that have been made to reach this point. It also provided a brief overview of some 3D programming concepts to familiarise the reader with various techniques. Each of the last three chapters investigated one of the APIs of interest. These sections outlined how the APIs worked; how they performed basic programming; and, then advantages and disadvantages.

OpenGL, the first API discussed, is one of the most popular APIs for 3D visualisation applications such as CAD and is widely accepted as the standard. It gained this popularity because it is supported by virtually all workstation suppliers and is available on most platforms through third party vendors. Most importantly perhaps, though, it is very simple to use providing a consistent interface, great versatility and it is network transparent. However, it requires hardware capabilities that are only now becoming commonly available in cheaper video cards. It provides only limited support for control and input functions and no facilities for extra features such as sound, virtual reality displays or force feedback input/output devices.

Direct3D and its family of related APIs within DirectX were introduced to enable fully featured world class games and interactive graphics development for the Windows platform. DirectX provides a device-independent access to all the hardware devices on the system such as 3D video-display hardware, sound hardware, as well as input and force feedback devices. It is a highly versatile API, providing very fine tuned control over every feature provided. Its greatest advantage over OpenGL is that by not being a standard it has not only been keeping up with hardware development but pushing the frontiers in the

introduction of new features. The primary problem with Direct3D and DirectX is that it is extremely complicated, requiring significant development to perform simple tasks, and because it is constantly updated developers need to be constantly retrained with each new release.

Java 3D is one of the newest APIs on the market and can be used for writing stand-alone 3D graphics or Web-based 3D applets. It is similar to some 3D-Engines on the market, in that it abstracts away from the complexity of the underlying complexities found in low-level APIs. This makes it more accessible to new developers that are unfamiliar with 3D programming. Its greatest power comes from its Java foundation, in that it provides a platform independent development environment and is tailored towards networked environments. It also provides extensive support for a number of virtual-reality-based devices and provides access to many advanced effects. While Java3D is an interesting inclusion into the market, it is a long way from becoming a leading development environment. This is primarily because its abstraction limits its versatility and its still being too slow for the more intense interactive applications.

All three APIs described in this paper have been shown to have areas in which they are particularly adept. For instance, since its introduction, DirectX has become the leading interface for game development, although most games provide users the ability to switch to OpenGL. While OpenGL still maintains a strong hold on the CAD and CAE applications, DirectX's introduction of more OpenGL like code is making some inroads into this market. Java 3D, while still developing its position in the market, is well placed for interactive applications developed for the Web.

When developing an application the developer needs to select the API that will best serve its purpose. For instance, if high versatility is required such as being able to finely control all aspects of the rendering pipeline then DirectX should be used but if multi-platform and maintainable code is required then OpenGL is better suited. Finally, if the application is web-based and not too intensive then Java 3D provides a very simple interface that should serve your needs.

# Bibliography


? Akeley, Kurt., "How OpenGL differs from the IRIS GL", last viewed 21st July 2000, http://reality.sgi.com/opengl/ogl_vs_igl.html.

? Angel, Edward., "Interactive Computer Graphics: A top-down approach with OpenGL", Addison-Wesley, Reading, Massachusetts, 1997.

? anon (a), *What Price Standards? Sometimes, Standards Stand in the Way of Innovation*, Microprocessor Report, v11 i11 pNA, 23rd August, 1999.

? anon (b), "Java 3D API 1.1 Performance Guide", last viewed 4th Aug, 2000. http://java.sun.com/products/java-media/3D/collateral/j3d_perfguide.html.

? Beier, Ekkehard., *Issues on Hierarchical Graphical Scenes*, Programming Paradigms in Graphics: Proceedings of the Eurographics Workshop in Maastricht, The Netherlands, published by SpringerWienNewYork, September 2-3, 1995.

? Bryson, Travis., *Exploring the Java 3D API*, UNIX Review's Performance Computing, v17 i4 p28(1), April 1999.

? Carson, George S., *Standards Pipeline: The OpenGL Specification*, Computer Graphics, A publication of ACM SIGGRAPH, vol 31 number 2, May 1997.

? Day, Bill., *3D Graphics Programming in Java, Part 1: Java 3D*, JavaWorld: Fueling Innovation, Dec 1998, last viewed 4th Aug, 2000, http://www.javaworld.com/javaworld/jw-12-1998/jw-12-media_p.html.

? Day, Bill., *3D Graphics Programming in Java, Part 2: Advanced Java 3D*, JavaWorld: Fueling Innovation, Jan 1999, last viewed 4th Aug, 2000, http://www.javaworld.com/javaworld/jw-01-1999/jw-01-media_p.html.

? De Goes, John., "3D Game Programming with C++", Coriolis Group Books, Scottsdale, Arizona, 1996.

? Foley, James D., van Dam, Andries., Feiner, Steven K., and Hughes, John F., "Computer Graphics: Principles and Practice", Second edition, Addison-Wesley, Reading, Massachusetts, 1996.

? Glaeser, Georg., "Fast Algorithms for 3D-Graphics", Springer-Verlag, New York, 1994.

? Hall, Roy., and Forsyth, Danielle., "Interactive 3D Graphics in Windows", Springer-Verlag, NewYork, USA, 1995.

? Hearn, Donald., Baker, M.Pauline., "Computer Graphics: C version", Second Edition, Prentice-Hall Upper Saddle River, New Jersey, 1997.

? Hearn, Donald., Baker, M.Pauline., “Computer Graphics”, Prentice-Hall Intenational Editions, 1986.

? Jacob, Karl., *Why Java and VRML: Understanding why Java and VRML are uniquely suited to each other*, JavaWorld: Fueling Innovation, March 1996, last viewed 4th August 2000. http://www.javaworld.com/javaworld/jw-03-1996/jw-03-javavrml_phtml.

? King, Andrew., “Software Considerations Implementing Virtual Realities”, University of Tasmania, 1992.

? Kovach, Peter, J., “Inside Direct 3D”, Published by Microsoft Press, March 2000.

? Microsoft Corporation, Inc., “DirectX 6.0 SDK”, Microsoft Corporation, Inc., 1999.

? Microsoft Corporation, Inc., “DirectX 7.0 SDK”, Microsoft Corporation, Inc., 1999.

? Microsoft Corporation, Inc., “Win32 SDK”, Microsoft Corporation, Inc., 1995.

? Mohan, Subra., “The Fourth Generation of 3D Graphics APIs has Arrived!: A new Generation of 3D API Emerges”, published by Sun Microsystems, 1998.

? Ribarsky, William., *The Times are A-Changing: PC Graphics Moves In*, IEEE Computer Graphics and Applications, published by the IEEE Computer Society, vol 8 number 3, May/June 1998.

? Salvator, Dave., *Full-Scene Anti-aliasing: Worth the Penalty? Is it time for a new 3D card?*, Computer Gaming World, p114, August 2000.

? Segal, Mark., and Akeley, Kurt., “The Design of the OpenGL Graphics Interface”, published by Silicon Graphics, Inc., Mountain View, California, USA, 1994.

? Segal, Mark., and Akeley, Kurt., “The OpenGL Graphics Interface”, published by Silicon Graphics, Inc., Mountain View, California, USA, 1993.

? Segal, Mark., and Akeley, Kurt., “The OpenGL Graphics System: A Specification”, Version 1.2.1, published by Silicon Graphics, Inc., 1999.

? Shreiner, Dave. and the OpenGL Architecture Review Board, “OpenGL Reference Manual: The Official Reference Documents to OpenGL”, Version 1.2, 3rd Edition, Addison-Wesley Pub Co., 1999.

? Sowizral, Henry A., “The Java 3D API: Technical White Paper”, published by Sun Microsystems, July 1997.

? Sowizral, Henry A., and Deering, Michael F., *The Java 3D API and Virtual Reality*, Projects in VR, May/June, 1999.

? Sowizral, Henry A., Rushford, Kevin., and Deering, Michael F., "The Java 3D Specification", version 1.2, published by Sun Microsystems, 2000.

? Strothotte, Christine., and Strothotte, Thomas., "Seeing Beiween the Pixels: Pictures in Interactive Systems", Springer-Verlag, NewYork, USA, 1997.

? Torborg, Jay., and Kenworthy, Mark., "A Look at DirectX 6.0, Fahrenheit, and the Future of Microsoft's Multimedia APIs: An Interview with DirectX Chief Jay Torborg and Group Program Manager Mark Kenworthy", MSDN Interview, Sept 4th, 1998, last viewed 17th July, 2000. http://msdn.microsoft.com/library/welcome/dsmsdn/msdn_torborg.htm.

? White, Josh., "Designing 3D Graphics: How to Create Real-Time 3D Models for Games and Virtual Reality", John Wiley and Sons, Inc, New York, USA, 1996.

? Woo, Mason. and the OpenGL Architecture Review Board, "OpenGL® 1.2 Programming Guide: The Official Guide to Learning OpenGL", Version 1.2, 3rd Edition, Addison-Wesley Pub Co., 1999.

? Zukowski, John., and Papageorge, John., "Big Splash: Java Technology and the Virtual Fish Tank", last viewed 4th August, 2000. http://java.sun.com/features/1998/11/fishtank.html.